Poznámky z Úvodu do databázových systémov - materiál na štátnice

Peter Csiba, petherz@gmail.com, https://github.com/Petrzlen/fmfi-poznamky

17.06.2012

Obsah

Úvod

Text je poznámkami k oficiálnym štátnicovým otázkam a boli spísané počas učenia sa na ne. Poznámky sa nesnažia ísť do hĺbky (na to je H.Garcia-Molina, Ullman a Wikipédia). Naopak, snažia sa priniesť intuitívnu predstavu o algoritmoch a pojmoch a dávať ich do súvisu.

Poznámky sú organizované podľa štátnicových otázok, snažia sa minimalizovať omáčku a nevysvetlujú a neuvádzajú do problematiky. Text je určený čitateľom, ktorý sa už s hlavnými pojmami stretli. Autor považuje všetky otázky za rovnako dôležité a odporúča v prípade slabšieho pochopenia samostatne vypracovať niektoré staré písomky.

Autor absolvoval základný test predmetu Úvodu do databázových systémov s hodnotením A, a má základné skúsenosti s administráciou a návrhom databáz. Napriek tomu autor neručí za kvalitu a úplnosť textu a čitateľov aj preto autor prudko odporúča pozrieť si aj iné zdroje. Uvedieme citát[1] "Tieto slajdy sú sprievodcom pri prednáške, nie sú myslené ako náhrada prednášky či nebodaj knihy. K príprave na skúšku nestačí len prečítať slajdy", niečo také sú aj tieto poznámky.

Nakoniec poznamenajme, že autor sa snažil písať pravdu a len pravdu, keďže jeho odpoveď na záverečných skúškach vychádza z tototo materiálu. Ak čitateľ chce prispieť ku kvalite textu, nech autorovi napíše a ten mu udelí prístup do repozitára.

P.S. Autor zistil, že názvoslovie pijan, ľúbi, krčma, alkohol použil už napríklad Ullman.

Úvod a motivácia databáz

Účel.

- Zobrazenie vybranej časti reality v počítačí.
- Uchovávanie informácií v konzistentnom stave a ich pridávanie.
- Poskytovanie informácií (dotazy, reporty - periodické správy).
- Ochrana informácií pred zničením a neoprávneným prístupom.

Charakteristiky DBMS (Database Management System).

- Dáta majú štruktúru, ktorá sa zriedka mení, objem dát je veľký.
- Dáta nie sú uložené na užívateľovom počítači, pre prístup k dátam sa využíva počítačová sieť (klient-server).
- Dotazy sú zložité.
- Množstvo užívateľov pristupuje k dátam súčasne.
- Vyžaduje sa vysoká priepustnosť.
- Vyžaduje sa vysoká odolnosť voči poruchám.
- Vyžaduje sa vysoký stupeň bezpečnosti.
- Prístup koncových užívateľov k dátam musí byť jednoduchý(API, GUI).

[1] Dr. Tomas Plachetka, Uvod do databazovych systemov 2011/2012 Zima

Porovnanie dotazov v dátových modeloch. Vo všeobecnosti sa snažíme zbaviť všeobecných kvatifikátorov pomocou pravidla $\forall P \rightarrow \neg\exists\neg P$.

Máme tabuľky:

- studenti(Student, Skupina)
- rozvrh_skupiny(Skupina, Miestnost, Cas)
- rozvrh_ucitelia(Ucitel, Skupina)
- *Hovorovo.* Treba nájsť (všetky) štvorice [Student, Miestnost, Cas, Ucitel] také, že ten učiteľ učí toho študenta v tej miestnosti a tom čase.
- *Predikátový kalkul.*

  {[Student, Miestnost, Cas, Ucitel]: ∃ Skupina (Student, Skupina) ∧ rozvrh_skupiny(Skupina, Miestnost, Cas) ∧ rozvrh_ucitelia(Ucitel, Skupina)}
- *Relačná algebra.*

$$\Pi_{student,miestnost,cas,ucitel}(studenti \bowtie rozvrh_skupiny \bowtie rozvrh_ucitelia)$$

- *Datalog (Prolog).*

```
rozvrh_studenti(Student, Miestnost, Cas, Ucitel) <--
\begin{verbatim}
  studenti(Student, Skupina),
  rozvrh_skupiny(Skupina, Miestnost, Cas),
  rozvrh_ucitelia(Učitel, Skupina).

? rozvrh_studenti(S, M, C, U)
```

- *SQL.*

```
SELECT S.student, RK.miestnost, RK.cas, RU.ucitel
FROM studenti S, rozvrh_skupiny RK, rozvrh_ucitelia RU
WHERE S.skupina=RK.skupina AND S.skupina=RU.skupina
```

1 Dátové modely

Pozri Wikipédiu.

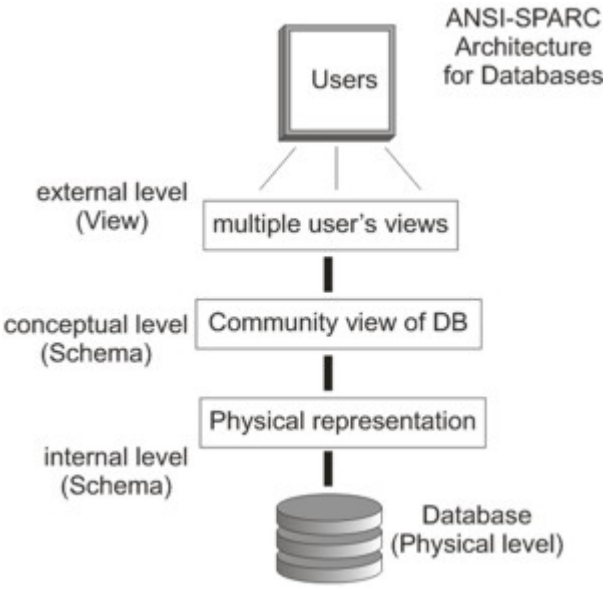
Používané dátové modely.

- Entitno-relačný. Pozri 1.2.
- Relačný. Pozri 2 a 4.
- Navigačný (XML). Stromová štruktúra, dotazovací jazyk XPath a XQuery.
- Objektový. Niečo ako objekty v programovaní. Objekty okrem zapuzdrenia dát špecifikujú prístup k nim a k ich vzťahom, je možné s nimi programaticky narábať. Napríklad DOM (Document Object Model).

1.1 Trojschémová architektúra (ANSI sparc)

Úrovne.

- Konceptuálna úroveň (všeobecná databázová schéma).
- Interná úroveň (špecifikácia polí, kľúčov, indexov, ...).
- Fyzická úroveň. Algoritmy a dátové štruktúry, využtie diskového priestoru. Pozri 10.



Vlastnosti.

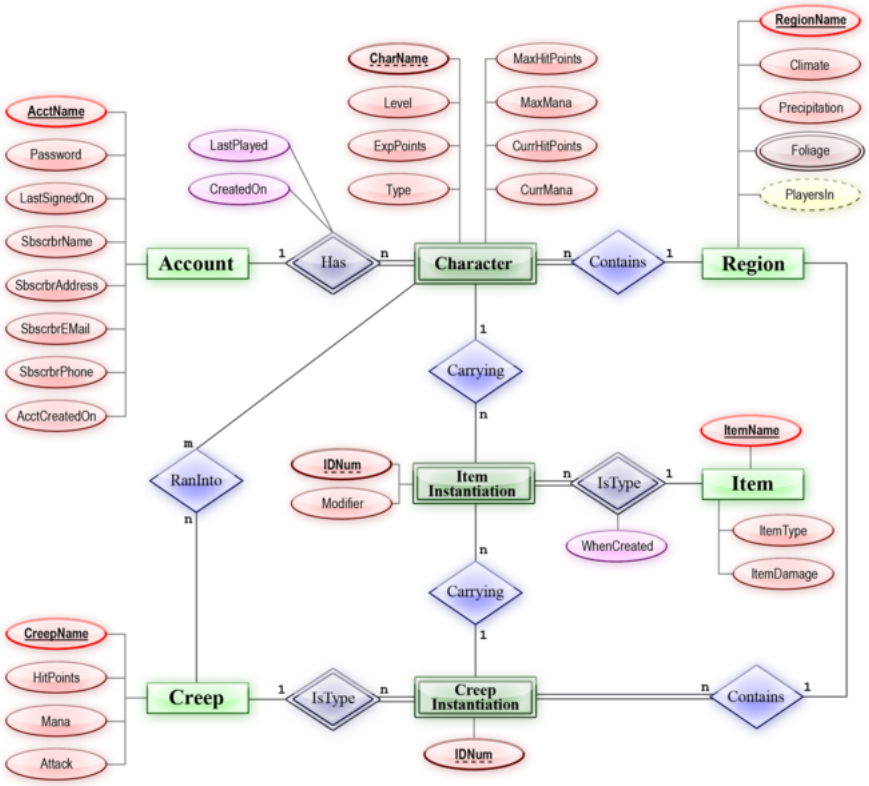
- Nezávislý pohľad užívateľov. Každý môže použiť svoj dotazovací jazyk.
- Užívateľ nevidí fyzickú organizáciu dát. Tá sa môže zmeniť, bez toho, aby to užívateľ postrehol.

1.2 Entitno-relačný model

Máme entity (tabuľky, polia, ...) a vzťahy medzi nimi.

Typy vzťahov.

- One-to-one (1-1). Napríklad hlava a telo.
- One-to-many (1-n). Napríklad vozidlo a kolesá.
- Many-to-many (n-m). Napríklad kupujúci a produkt (táto dvojica nemusí byť unikátna).



1.3 Relačný model, relačná algebra

Pozri 2 a 4.

1.4 Negácia a rekurzia v relačnej algebre

Pozri 4.

1.5 Súvis relačnej algebry s inými dotazovacími jazykmi

Pozri 4.5.

2 Relačný kalkul

Deklaruje výroky, ktoré ohraničujú výsledok dotazu.

Použité pojmy. Je odporúčané vedieť matematickú logiku.

- n-árna relácia (konečná, nekonečná). Usporiadané n-tice. Karteziánsky súčin.
- Atribúty - zložky n-tíc. Domény (typy atribútov) - množiny karteziánskeho súčinu.
- Predikát P prislúchajúci relácií R- zobrazenie $R_P : R \mapsto \{\text{true, false}\}$, pričom $R_P(A) = true \leftrightarrow A \in R$, kde A je usporiadaná n-tica. V praxi medzi tým nerozlišujeme.

Dotaz. Dotazy v relačnom kalkule sú logické formuly predikátovej logiky prvého rádu. Formula by nemala mať voľné premenné. Výsledkom sú prvky relácie, ktoré spĺňajú formulu.

2.1 Predikátová interpretácia relačnej algebry

??? Asi ide o prepis symbolov relačnej algebry do predikátovej logiky.

2.2 Negácia, doménovo nezávislé a bezpečné formuly

Príklad dotazov.

1. prvocisla1: $\{Z : (\forall X)(\forall Y)(krat(X,Y,Z) \rightarrow X = 1 \wedge Y = 1)\}$.
2. zlozene_cisla: $\{Z : (\exists X)(\exists Y)(krat(X,Y,Z) \wedge X \neq 1 \wedge Y \neq 1)\}$.
3. prvocisla2: $\{Z : (\exists X)(\exists Y)(krat(X,Y,Z) \wedge Z \notin zlozene\_cisla)\}$.
4. prvocisla3: $\{Z : Z \notin zlozene\_cisla\}$.

Tretiu formulu (reláciu, resp. množinu) sme získali dvojitou negáciou prvej a pravidlom $P \rightarrow Q \Leftrightarrow \neg P \vee Q$ (medzi krok je druhá formula).

Formuly pre prvocisla1 a prvocisla3 sú *nebezpečné*, keďže ich výsledom je napríklad aj "petrzlen". V prvom prípade je implikácia pravdivá (lebo ľavá strana je nepravdivá) a v druhom prípade je to zjavné, keďže "petrzlen" nie je zložené číslo. Správna je len formula prvocisla2.

Množina faktov (nazývaná aj *extenzionálna databáza (EDB)*) je konečná množina, na ktorej je predikát v dotaze pravdivý (to zrejme neplati pre prvocisla1 a prvocisla3, ale platí pre prvocisla2 a zlozene_cisla). Neformálne, *bezpečný dotaz* je taký, v ktorom sa každá premenná vyskytuje v aspoň jednom EDB v pozitívnom kontexte. Tj. chceme, aby sa premenná nevyskytovala v negovanej množine faktov (výroku). Napríklad v prvocisla3 je Z v negatívnom kontexte.

2.3 Relačný kalkul (doménový)

Myslím, že je to popísané v predošlých sekciách. Pripomeňme, že *domény* sú typy atribútov - množiny karteziánskeho súčinu.

2.4 Súvis relačného kalkulu s inými dotazovacími jazykmi

Základná intuícia. Celkom dobre sa doň prekladá z hovorového a jendoducho sa z neho konštruuje datalog, resp. relačná algebra. V praxi nie je použiteľný, keďže ide o teoretický model (s potenciálne nekonečnými reláciami).

3 Datalog

Datalog je dotazovací jazyk s pravidlami určený pre deduktívne databázy, ktorý je syntakticky podmnožinou Prologu. Je založený na predikátovej logike prvého rádu (neumožňuje kvantifikovať množiny) a umožňuje rekurziu.

Odporúčame si pozrieť slide-i Plachetku pre ďalšie príklady dotazov. Ideálne si niektoré úlohy treba vyskúšať v Datalógovom interpretery (tak sa to učí najlepšie).

3.1 Syntax a sémantika Datalogových programov

Syntax. Program pozostáva z množiny pravidiel - *implikácií* - *predikátov*, ktoré majú *hlavu* a *telo*[2] (ľavá a pravá strana implikácie, resp. *premisa* a *dôsledok*). Napríklad:

```
1. krat(1, 1, 1) <-- TRUE.
2. krat(1, 1, 1).
3. zlozene_cislo(Z) <--
  krat(X, Y, Z),
  not X = 1,
  not Y = 1.
4. prvocislo(Z) <--
  krat(X, Y, Z),
  not zlozene_cislo(Z)
5. prvocislo(Z) <--
  krat(_, _, Z),
  not zlozene_cislo(Z)
```

Predikáty 1 a 2 a aj predikáty 4 a 5 sú sémanticky ekvivalentné. Premenné na ľavej strane sú kvantifikované $\forall$ a premenné na pravej strane $\exists$. Čiarka medzi atómami má význam AND. Bodka označuje koniec predikátu. *Podcieľmi* nazývame atómy tela rozdelené čiarkami, napríklad `{krat(_, _, Z)}` a `not zlozene_cislo(Z)` sú podcieľmi `prvocislo(Z)`. V 5 používame *anonymné premenné* "_". Ich sémentika je rovnaká, ako keby sa použila ľubovoľná nepoužitá premenná. Poznmenajme, že v tele pravidla sú premenné implicitne kvantifikované $\exists$.

Alebo (disjunkcia) sa vyjadruje:

```
colour(F) <-- F = black
colour(F) <-- F = white
```

Ďalšie syntaktické pravidlá.

- Argumentmi (hláv) predikátov môžu byť len jednoduché atribúty. Nemôžeme `zlozene_cislo(f(f(Z,5)) <--` .
- (Tvrdí Wiki, nie Plachetka). Každá premenná v hlave predikátu sa musí vyskytovať aj v nie negovanom podcieli predikátu (súvisí s bezpečnosťou).

Sémantika. Je založená na relačnom kalkule (a matematickej logike). Niečo sa spomenulo vyššie, doplňujeme v ďalšej sekcií.

Agregácia (subtotal).

```
objednava_od_hp(K, V) <--
objednavky(K, V), dodava(hp, V, _, _).
answer(K, P) <--
subtotal(objednava_od_hp(K, V), [K], [P = count(V)]).
```

Príkaz `subtotal` má tri parametre. Prvým je predikát, druhým je *grupovací atribút* a tretím *agregovací atribút*. Viac v 4.3.

Podobne sa dá definovať subtotal aj pre relačný kalkulus.

[2] Nie pätu. Poznámka autora.

3.2 Súvis s relačným kalkulom

Napríklad

```
3. zlozene_cislo(Z) <--
  krat(X, Y, Z),
  not X = 1,
  not Y = 1.
```

je sémanticky ekvivalentným programom k

$$zlozene\_cisla : \{Z : (\exists X)(\exists Y)(krat(X, Y, Z) \wedge X \neq 1 \wedge Y \neq 1)\}.$$

Existuje algoritmický predklad medzi týmito dvoma modelmi. Algoritmus, ktorý prepisuje relačný kalkul do Datalogu pozostáva z nasledujúcich krokov (treba ich vedieť použiť):

1. Premenovanie premenných, lokalizácia kvantifikátorov. Vlastne veta o variantoch z matematickej logiky.
2. Eliminácia všeobecných kvantifikátorov a implikácií. Použijeme pravidlá $(\forall x)A(x) \leftrightarrow \neg(\exists x)\neg A(x)$ a $(A \rightarrow B) \leftrightarrow (\neg A \vee B)$.
3. Definícia pomocných predikátov. Chceme sa vyhnúť sekvencií $\neg(\exists X)povodny\_predikat(X)$. Definujeme preto pomocné predikáty:

$$\neg novy\_predikat$$
$$novy\_predikat \leftarrow (\exists X)povodny\_predikat(X).$$

4. Bezpečnosť pomocných predikátov. Pridáme k premenným ich dómeny:

$$novy\_predikat \leftarrow domena(X) \wedge (\exists X)povodny\_predikat(X).$$

5. Prepis do pravidiel Datalogu. Len zmeniť syntax:

```
answer <-- not novy_predikat.
novy_predikat <-- povodny_predikat(X).
```

Poznamenajme, že podobne existuje algoritmus, ktorý prepisuje datalogove dotazy do SQL (tiež treba vedieť použiť).

3.3 Výpočet dotazu na Datalogový program

Datalog počíta *minimálny model* programu (minimálne relácie, pre ktoré je daný program splnený). Inými slovami, Datalog vracia najmenšiu množinu, pre ktorú sú splnené všetky potrebné predikáty, na ktorých závisí dôsledok (dotaz). Tj. neuvažuje také výsledky, pre ktoré sú predpoklady nesplnené. Napríklad program:

```
colour(black).
colour(white).
? colour(C).
```

by mohol vrátiť aj "mrkva", lebo nič tomu neodporuje.

Minimálny model existuje pre každý *bezpečný program* (ktorý sa definuje prakticky rovnako, ako v relačnom kalkule).

3.4 Negácia

Russelov paradox.

```
man(barber).
man(mayor).
shaves(barber, X) <-- man(X), not shaves(X, X).
? shaves(barber, barber)
```

Paradox nenastáva, keďže pravidlo je definované ako implikácia. Výsledok nemôže byť FALSE, lebo by predpoklad bol splnený. Ak je výsledok TRUE, tak `man(X), not shaves(X, X) --> shaves(barber, X)` platí.

3.5 Bezpečnosť Datalogových programov

Bezpečný predikát je taký, ktorého všetky premenné vyskytytujúce sa v hlave, sú viazané v EDB.

Bezpečný program má všetky svoje predikáty bezpečené.

(Zopakujeme) Minimálny model existuje pre každý bezpečný program (ktorý sa definuje prakticky rovnako, ako v relačnom kalkule).

Všeobecne platí, že anonymné premenné v negovaných EDB podcieľoch nespôsobujú problém s bezpečnosťou.

Pán Šturc definujeme slabšie podmienky.

4 Relačná algebra

Hovorí AKO sa má niečo vypočítať. Operuje len nad konečnými množinami (aby neboli problémy s negáciami). Formalizuje SQL. Viac na Wikipédií - náhodny poznatok odtiaľ: Reflexívno tranzitívny uzáver sa nedá relačnou algebrou vyjadriť.

V tejto sekcií budeme implicitne veľkými písmenami značiť m-tice atribútov nad n-reláciami r ako $X = (X_1, X_2, \ldots, X_m)$ (nemusí $n = m$).

4.1 Operátory relačnej algebry

Operátory.

- Projekcia Π. Nech r má atribúty Y, potom $\Pi_X(r)$ vznikne kopírovaním riadkov r, pričom sa skopírujú len tie atribúty, ktoré sú v X. Treba eliminovať potenciálne duplicity.
- Selekcia σ. Nech c je podmienka a X atribúty r, potom $\sigma_{c(X)} = \{X : r(X) \wedge c(X)\}$.
- Kartézsky súčin $\times$. Relácie r_1 a r_2, atribúty X, Y pričom $X \cap Y = \emptyset$, potom $r_1 \times r_2 = \{[X, Y] : r_1(X) \wedge r_2(Y)\}$.
- Join (theta-join). Podmienka c, relácie r_1 a r_2, atribúty X, Y pričom $X \cap Y = \emptyset$, potom $r_1 \bowtie_{c(X,Y)} r_2 = \{[X, Y] : r_1(X) \wedge r_2(X) \wedge c(X, Y)\}$.
- Premenovanie $P_{r_2(Y)}(r1)$ atribútov relácie.
- Natural join $\bowtie$. Nech r_1 je typu[3] $r_1(X, Z)$ a r_2 je typu $r_2(Y, Z)$, pričom Z sú spoločné atribúty r_1 a r_2. Potom $r_1 \bowtie r_2 = \{[X, Y, Z] : r_1(X, Z) \wedge r_2(Y, Z)\}$, pričom spoločné atribúty (atribúty s rovnakým menom) sú testované na rovnosť a sú kopírované iba raz. Natural join sa dá vyjadriť projekciou a theta-joinom.

[3] Všetky atribúty r_1 je zjednotenie X a Z

- Tradičné množinové $\cap, \cup, \ldots$.
- Eliminácia duplikátov $r_2 = \Delta(r_1)$.

Optimalizácia výpočtu. Použitie operátorov sa zvykne zapisovať v strome, kde operátory sú vrcholy a argumenty (relácie) hrany. Pre jeden výraz môže existovať viacero výpočtovo ekvivalentných stromov. Zopár pravidiel na konštrukciu optimálneho:

- Selekciu urob čo najskôr.
- Vyhýbaj sa kartézskym súčinom, nahraď ich joinom. Ak nasleduje selekcia, pridaj ju do joinovacej podmienky.
- Postupnosť unárnych operácií (selekcie a projekcie) spoj do jednej operácie a zviaž s nasledujúcou operáciou.
- Výsledok opakovaného podvýrazu ulož, ak je veľkosť tabuľky malá, tak ju materializuj.
- Nájdi optimálne poradie joinov technikou dynamického programovania. Poradí pre N tabuliek je zhruba 2^N (binárne stromy).

4.2 Multimnožinová interpretácia relácií

Plachetka: "Relačná algebra počíta nad multimnožinami". Na Wikipédií som to nenašiel explicitne spomenuté a takisto sme operátory definovali pre množiny. V každom prípade je možné jednoduchým spôsobom rozšíriť definície na *multimnožiny*[4]. Kto nevie, Wikipédia.

Niektoré základné vlastnosti operácií sa nezachovávajú, ako napríklad $S \cup S = S$.

4.3 Grupovanie a agregácia

(Skopírovaný slide.)

Atribúty. O každom atribúte pre operátor agregácie platí $x \in X$ platí buď

- x je atribútom r_1 (v tom prípade atribútu X hovoríme *grupovací atribút*), alebo
- $x = AGG(Y)$, kde Y je atribútom r_1 a AGG je nejaká agregačná funkcia, ktorá zo stĺpca Y vyrába jednu hodnotu (v tom prípade atribútu x hovoríme *agregovaný atribút*) V SQL je $AGG \in$ {SUM, COUNT, AVG, STDEV, MAX, MIN}.

Operátor. $\Gamma_X(r_1)$

1. najprv vyrobí z relácie r_1 skupiny, pričom riadky v každej zo skupín majú rovnaké hodnoty grupovacích atribútov
2. potom vypočíta agregované atribúty pre každú skupinu.

Výsledkom je tabuľka, v ktorej každej skupine prináleží jeden riadok.

Implementuje sa projekciou (multimnožinovou) a vyhodením duplikátov.

[4] Rovnaký prvok sa môže vyskytnúť viackrát

	SELECT $< S\_attr >$	5
	FROM r_1, r_2, ..., r_n	1
SQL:	WHERE $< w\_cond >$	2
	GROUP BY $< G\_attr >$	3
	HAVING $< h\_cond >$	4

Relačná algebra:

$$\Pi_{S_attr}(\sigma_{h_cond}(\Gamma_{G_attr,agg(Attr)}(\sigma_{w_cond}(r_1 \times r_2 \times \ldots \times r_n))))$$

.

4.4 Rekurzia, výpočet pevného bodu

Podľa Wiki základná relačná algebra nevie rekurziu (to je celkom zrejmé). Plachetkov slide:

Operátor *minimal fixpoint* (tranzitívny uzáver) $\Phi(< vyraz >)$ iteruje výraz, kým sa relácie použité vo výraze menia (t.j. kým sa mení obsah aspoň 1 relácie výrazu)

Príklad:

ancestor := parent(X, A)
ancestor := ancestor $\cup\Pi_{X,A}$(ancestor $\bowtie$ parent)
ancestor := ancestor $\cup\Pi_{X,A}$(ancestor $\bowtie$ parent)
...

sa zapíše ako

ancestor := $\Phi(\cup\Pi_{X,A}$(ancestor $\bowtie$ parent))

Výpočet pevného bodu patrí do predmetu Základy teórie programovania (blok B1). Naivný algortimus:

```
answer = {};
do
{
answer = iteration_step(Φ1);
answer = iteration_step(Φ2);
...
answer = iteration_step(Φn);
} while answer changed; /* anywhere inside this loop */
```

Oveľa viac v slide-och z roku 2011. Spíšem, ak mi zostane čas.

4.5 Súvis relačnej algebry s inými dotazovacími jazykmi

SQL bolo inšpirované relačnou algebrou.

5 Jazyk SQL (Structured Query Language)

SQL je jazykom pre prístup k relačným databázou navrhnutým na základe relačnej algebry a relačného kalkulu. Predpokladáme, že väčšina pozná (alebo si aspoň myslí, že pozná). Takže len stručne. Kto nevie, tak Wikipedia.

5.1 Programovanie v SQL (DDL, DML)

SQL je typový jazyk.

Poznamenajme, že SQL používa trojhodnotovú logiku: {TRUE, FALSE, NULL} a tabuľky nie sú množinami, ale zoznammi (rovnaká n-tica - riadok - sa môže vyskytovať viackrát). Obe sú v spore s relačnou algebrou.

Ďalej poznamenajme, že tabuľka nemusí byť materializovaná, môže byť dočasná alebo trvalá.

DDL (Data Definition Language). Vytvára (mení, maže) štruktúru dát, relácie, indexy, užívateľov (a prístupové práva), triggery, ...

DDL a DML sú podmnožinami jazyka SQL.

- CREATE. Najčastejšie CREATE TABLE.
- ALTER.
- RENAME.
- DROP. Navždy vymazať.
- TRUNCATE. Vyprázdniť.

DML (Data Manipulation Language). Slúži na prístup, pridávanie, zmenu a zmazanie dát.

- SELECT.
- INSERT.
- UPDATE.
- DELETE. Navždy vymazať.
- MERGE. Vyprázdniť.

Kľúčové slová pre SELECT.

- FROM. Z tabuľky.
- WHERE. Kde podmienka.
- `[FULL, LEFT, RIGHT] [INNER, OUTER] JOIN`. Spájanie viacerých tabuliek podľa podmienky. Odporúčame si pozrieť grafické vysvetlenie. Najbežnejší je LEFT INNER JOIN (resp. len JOIN). [FULL, LEFT, RIGHT] hovorí, z pohľadu ktorej JOINujeme a OUTER (default je INNER) hovorí, že zahŕňame aj riadok, ktorý nemá pár.
- CROSS. Kartézsky súčin.
- UNION. Množinové zjednotenie dvoch tabuliek. UNION ALL je spojenie.
- INTERSECT. Množinový prienik dvoch tabuliek.
- DISTINCT. Množinové správanie.
- EXISTS. Či SELECT vracia prázdny výsledok.
- ANY. Ľubovoľný vyhovujúci prvok z tabuľky. Použitie sa dá obísť, ide skôr o skratku.
- LIMIT. Obmedzenie počtu výstupných riadkov.
- ORDER BY. Usporiadanie výstupu.

Agregácia.

- GROUP BY. Agregácia.
- COUNT.
- SUM.
- AVG.

NULL. Poznamenajme, že hodnota NULL sa môže chovať kontraintuitívne. Napríklad:

```
SELECT r.C
FROM r
WHERE A < 1 OR A >= 1
```

nevráti riadky, kde $A = 0$.

Index. Uľahčujú vyhľadávanie, väčšinou sú implementované binárnymi vyhľadávacími stromami. Tj. namiesto vyhľadávania v čase $O(n)$ máme $O(logn)$.

5.2 Negácia a rekurzia v SQL

O negácií to isté ako o negácií v Datalogu.

Rekurzia v SQL je možná, napr. pomocou WITH, ktorý definuje dočasnú tabuľku. Rekurzia nastáva, keď sa odvolávame na tabuľku, ktorú práve definujeme. Koho viac zaujíma, môže si pozrieť praktický príklad.

Konstra rekurzívneho programu pre UNION ALL:

```
01. <typical_recursive_union> ::= WITH RECURSIVE <local_view_name>
02. "(" <alias_name_list> ")"
03. AS "(" <initial_query_specification>
04. UNION ALL
05. <recursive_query_specification> ")"
06. <outer_query_specification>
07. [ <order_by_clause> ]
08. [ <for_clause> ]
```

5.3 Súvis SQL s inými dotazovacími jazykmi

Bol postavený na relačnej algebre (pozri 4) a relačnom kalkule 2. Existujú priamočiare algortimické transformácie s Datalogom a relačnou algebrou (operátorový strom).

Konverzia z Datalogu. Existuje jednoduchý algortimus na preklad Datalogu do SQL, využíva konštrukcie na vytváranie dočasných tabuliek:

```
WITH r AS (SELECT ...),
SELECT ...
```

alebo

```
CREATE TEMPORARY TABLE r AS (SELECT ...);
SELECT ...
```

alebo permanentnú, možno nie materializovanú:

```
CREATE VIEW r AS (SELECT ...);
SELECT ...
```

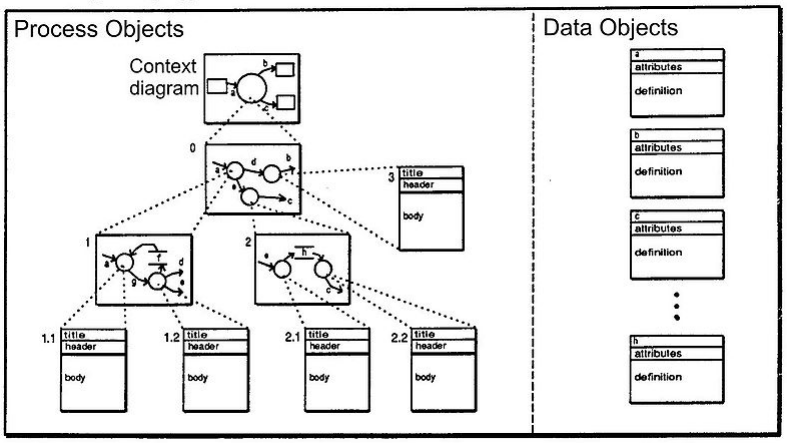
Hlavne CREATE VIEW si odporúčame pozrieť.

6 Teória navrhovania relačných báz dát

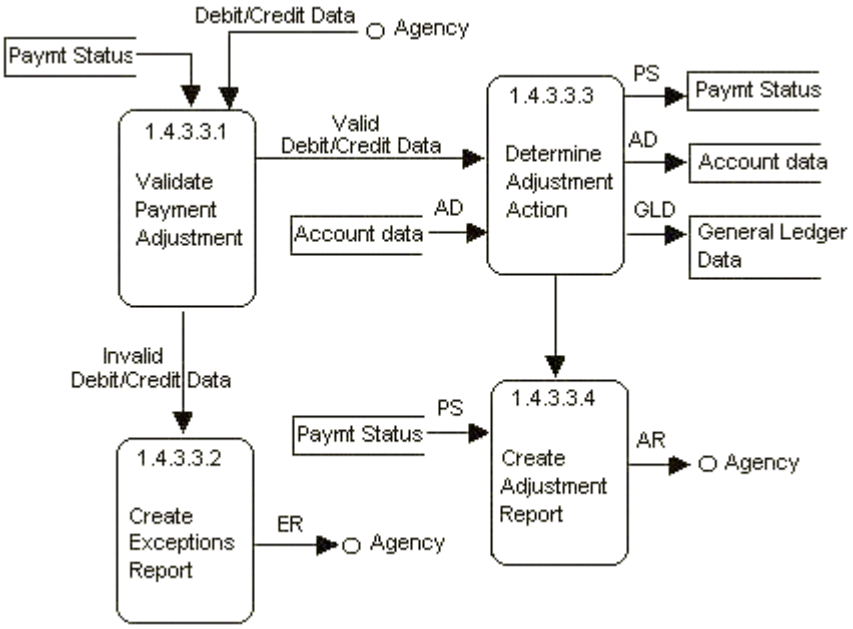
V tejto sekcií predpokladáme, že všetky spomínané databázy sú relačné.

Niektoré používané techniky.

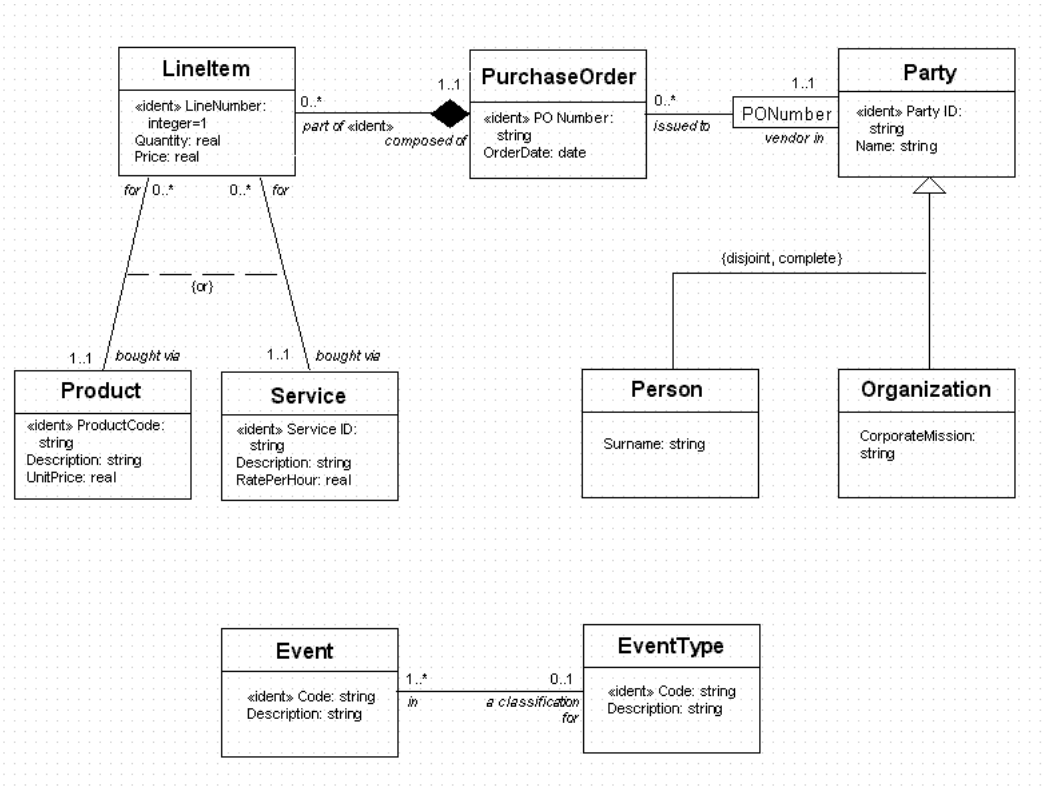
- SAD (Structured Analysis and Design). Zachytáva procesy aj dáta naraz. Technika z roku 1980.



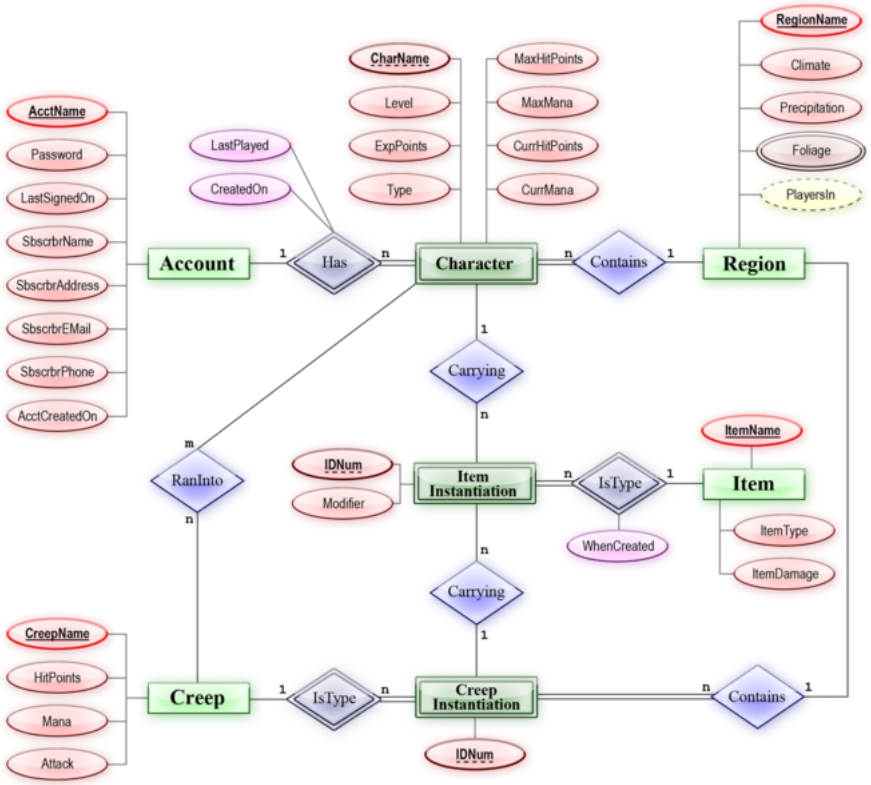
- DFD (Data-Flow Diagrams). Znnázorňuje, ako dáta prúdia cez systém, vstupy, výstupy.



- UML (Unified Modeling Language).S tým sa musel stretnúť každý, napríklad pri Design Patternoch.



- ER (Enitity Relationship). S tými sme sa už stretli, sú najvhodnejšie pre relačné databázy (ako už názov napovedá). Nevýhodou je, že nezachytávajú ako sa entity menia.



Existujú softwareové nástroje, ktoré výrazne uľahčujú ich tvorbu, resp. vedia konvertovať textovú schému na takéto diagramy (je ich dosť).

Kroky návrhu.

- *Koncepčný návrh.* Vytvorí sa diagram pre databázu a jej relácie. Zaujímajú nás entity a už menej ich atribúty a vôbec ich typy.
- *Logický návrh.* Typy atribútov, bezpečnostný model, optimalizácia (normalizácia, ...), kľúče. Predklad do SQL (ako normu pre relačné DB) môžeme považovať za takýto krok. Treba vedieť syntax `CREATE TABLE` a hlavne `FOREIGN KEY (id) REFERENCES (table)` a ktomu prislúchajúci `ON DELETE [CASCADE, SE` ktoré hovoria čo sa má spraviť, keď sa vymaže FOREIGN KEY.
- *Fyzický návrh.* Mapovanie návrhu na konkrétny DBMS (DB Management System).

Výsledkom sú vytvorené tabuľky, kľúče, indexy, constrainty, užívateľské kontá, procedúry vkladania/vynechávania dát, pohľady (VIEWs), prístupové práva atď.

Pravidlá dobrého návrhu.

- *Vyhýbať sa redundancií.* Ak sa dá niečo vypočítať, tak si to nepamätáme (jedine kvôli optimalizácií). Hlavným dôvodom je potenciálna nekonzistentnosť dát.
- *Vyhýbať sa slabým entity setom.* To sú také, ktoré nevedia odpovedať na všetky (zmysluplné) otázky. Štandardne *fan trap* - zlé poradie vzťahov entít v reťazi a *chasm trap* - chýbajúca závislosť (v trojici).

- *Nepoužívať entity set, ktorý sa dá nahradiť atribútom.*

Prudko odporúčame pozrieť si slide-i Plachetku pre názorné grafické ukážky chýb.

6.1 Funkčné závislosti

Úvod a označenia. Chceme odstrániť redundanciu a znej plynúcu potenciálnu nekonzistentnosť.

Ďalej budeme X_i označovať *inštanciu* množiny atribútov X relácie r. Inštanciou rozumieme riadok (prvok relácie) s projekciou na X. Ďalej, predikátom $r(X_i, Y_j, \ldots, Z_k)$ označujeme pre inštancie $(X_i, Y_j, \ldots, Z_k)$, či existuje spojená inštancia $(X_i, Y_j, \ldots, Z_k)$ v r. Spojenou inštanciou[5] rozumieme inštanciu S, ktorá má všetky atribúty relácie r. V prípade, že pre nejaké $X \cap Y \neq \emptyset$, tak inštancie X_i a Y_j musia mať na atribútoch $X \cap Y$ rovnaké hodnoty. Teraz sme vysvetľovali niečo, čo je takmer zrejmé, formalizmom, ktorý môže na prvé čitanie zmiasť.

Definícia. *Funkčná závislosť* $X \to Y$ práve vtedy keď

$$(\forall X_1)(\forall Y_1, Y_2)(\forall Z_1)(r(X_1, Y_1, Z_1) \wedge r(X_1, Y_2, Z_1) \Rightarrow (Y_1 = Y_2)).$$

Odteraz "$\Rightarrow$" implikácia a "$\to$" funkčná závislosť. Definujme makro $XY \equiv X \cup Y$.

Armstrongove axiómy.

- Reflexívnosť. $X \supseteq Y \Rightarrow X \to Y$.
- Rozšírenie (augmentation). $(\forall Z)(X \to Y \Rightarrow (XZ \to YZ))$.
- Tranzitívnosť. $(X \to Y \wedge Y \to Z) \Rightarrow (X \to Z)$.

Mali by ste vedieť dokázať (potom to ale nie sú axiómy, ale teorémy).

Armstrongové axiómy sú úplne: Funkčná závislosť $X \to Y$ sa dá odvodiť z F pomocou Armstrongovych axióm práve vtedy, keď $X \to Y$ je logickým dôsledkom[6] F.

Niektoré ďalšie vlastnosti.

- Union rule. $(X \to Y \wedge X \to Z) \Rightarrow (X \to YZ)$.
- Pseudotranzivita. $(X \to Y \wedge WY \to Z) \Rightarrow (WX \to WZ)$.

Mali by ste vedieť dokázať z Armstrongových. Všimneme si, že keď r nemá funkčné závislosti, tak r nemá redundantné dáta.

Uzáver množiny atribútov. (Skopírovaný slide.)

Nech X je množina atribútov a F množina funkčných závislostí. Potom *uzáverom množiny atribútov* X vzhľadom na F rozumieme množinu X^+ všetkých atribútov Y takých, že $X \to Y$ je (logickým) dôsledkom funkčných závislostí F.

```
X+ : = X;
repeat
for each U --> V in F do
if U in X+ then X+ := X+ union V;
while niečo sa pridalo do X+;
```

Optimalizácia: každá závislosť sa použije práve raz, po použití ju možno vynechať. (Teda uzáver sa počíta v lineárnom čase[7].)

[5] Definovali sme si len pre dočasné potreby, nevieme, či sa používa aj v odbornej literatúre.

[6] Učiteľ Matematickej logiky by sa nad týmto výrokom pozastavil. Logický dôsledok chápeme asi ako intuitívny pojem.

[7] Na F sme použili všetky Armstrongove axiómy (urobili sme niečo ako reflexívno tranzitívny uzáver).

Uzáver množiny funkčných závislostí. (Skopírovaný slide.)

Označme F^+ množinu všetkých funkčných závislostí, ktoré sú dôsledkom funkčných závislostí z F (t.j. ktoré sa dajú odvodiť z F použitím Armstrongovych axióm). Množinu F^+ budeme nazývať *uzáverom množiny funkčných závislostí* F.

Je dosť rozsiahla, chceme uvažovať len vhodnú podmnožinu. *Maximálna funkčná závislosť* - nemôže vynechať atribút z ľavej strany ani pridať atribút k pravej strane tak, aby sme neporušili jej platnosť.

6.2 Pokrytie a minimálne pokrytie množiny funkčných závislostí

Množina funkčných závislostí G *pokrýva* množinu funkčných závislostí F práve vtedy keď $G^+ \supseteq F^+$. Stačí testovať, či vieme odvodiť každú funkčnú závislosť z F pomocou G (polynomiálne).

Kanonická funkčná závislosť - je maximálna a na pravej strane má práve jeden atribút.

Minimálne pokrytie množiny funkčných závislostí - je pokrytie pozostávajúce len z kanonických funkčných závislostí.

Polynomiálny algoritmus na výpočet minimálneho pokrytia.

1. Nahraď $X \to Y$ množinou $\{X \to A,\ A \in Y,\ A$ je jednoduchý (jednotkový) atribút$\}$
2. Vynechaj všetky redundantné (odvoditeľné) atribúty na ľavých stranách $X \to A$ (každý atribút treba testovať práve raz).
3. Vynechaj všetky redundantné (odvoditeľné z ostatných) závislosti $X \to A$ (opakuj tento krok s redukovanou množinou funkčných závislostí, kým žiadna závislosť nie je redundantná, každú závislosť treba testovať práve raz). Viac v slide-och.

6.3 Nadkľúče a kľúče

Kľúč je (neformálne) minimálna množina atribútov, ktorá jednoznačne identifikuje entitu. Kľúčov môže byť viac. *Primárny kľúč* je niektorý z kľúčov a označuje sa počiarknutím atribútov, ktoré ho tvoria. Primátny kľúč by mal byť minimálny možný, odporúča sa použiť zástupcu *surrogate key* (napr. AUTO INCREMENT v SQL).

Formálne je *nadkľúč* množina atribútov K taká, že $K \to U$, kde U sú všetky atribúty relácie r. *Kľúčom* nazývame minimálny (v zmysle množinovej inklúzie) nadkľúč r (môže ich teda byť viacej).

Algoritmus na nájdenie kľúčov existuje len exponenciálny (určite ste ho vedeli na teste). Plachetka odporúča úplne preberanie kľúčov zhora (najdlhších), po najkratšie. Výpočet sa reprezentuje binárnym stromom, kde sa v každom vrchole množina aitribútov vetví podľa toho, či daný atribút zahrnieme, alebo nezahrnieme do potenciálneho kľúča (a podčiarkujeme si pritom atribúty, ktoré určite chceme zahrnúť).

7 Normálne formy

Tabuľky, ktoré nie sú v normálnych formách, môžu napríklad viesť k stratám závislosti pri vymazávaní alebo nekonzistentným dátam pri zmenách (viac v slide-och).

Relačná schéma. *Relačná schéma* je množina atribútov U a množina funkčných závislostí F, ktoré platia v relácií r. Tj, rozdeľujeme tabuľky na viacero tabuliek.

Dekompozícia relačnej schémy sú relácie r_i a funkčné závislosti F_i, pričom r_i vznikli projekciou r (a vymazaním duplikátov), $\Cup_i r_i = U$ a $F^+ \supseteq F_i$ (nepridali sa nové závislosti).

Bezstratová dekompozícia práve vtedy, keď $r = \Pi_{r_1}(r) \bowtie \cdots \bowtie \Pi_{r_n}(r)$. Všetky funkčné závislosti sa museli zachovať, inak by sa relácia r neobnovila. Poznamenajme, že r_i sme definovali ako relácie a v $\Pi_{r_1}(r)$ ich používame ako množiny atribútov.

Prvá normálna forma (1NF). (Historický problém) Atribúty definujeme nad doménami s atomickými hodnotami (nie napr. `enum`). V skutočnosti sa definuje trocha inak - pridávajú sa množinové vlastnosti riadkov a stĺpcov.

Druhá normálna forma (2NF). (Historický problém) Žiadny atribút v r_i nezávisí iba na časti kľúča r_i a každá z r_i je v 1NF. Tj., ak je kľúč (=minimálny nadkľúč) v tvare ABC, tak v tabuľke neexistujú funkčné závislosti, ktorých ľavé strany sú A, B, C, AB, BC alebo AC.

Employee	*Skill*	*CurrentWorkLocation*
Jones	Typing	114 Main Street
Jones	Shorthand	114 Main Street
Jones	Whittling	114 Main Street
Bravo	Light Cleaning	73 Industrial Way
Ellis	Alchemy	73 Industrial Way
Ellis	Flying	73 Industrial Way
Harrison	Light Cleaning	73 Industrial Way

Jediným kľúčom tabuľky je $(Employee, Skill)$, ale tabuľka nie je v 2NF, lebo existuje funkčná závislosť $Employee \to CurrentWorkLocation$.

7.1 3NF, BCNF

Tretia normálna forma (3NF). (Plachetka) Relačná schéma (r, F) je v 3NF práve vtedy, keď je v 2NF a pre každú (platnú) funkčnú závislosť $X \to Y$ platí, že buď X je nadkľúč v r, alebo Y je časťou nejakého kľúča v r. (Wikipédia, silnejšia podmienka) Relačná schéma (r, F) je v 3NF práve vtedy, keď je v 2NF a pre každú (platnú) pre každý atribút U_i platí, že sa buď vyskytuje v šavej strane nejakého kľúča r, alebo je odvoditeľný z každého nadkľúča r.

Testovanie 3NF je NP ťažké, ale jej konštrukcia je polynomiálna.

Ak to správne chápem, tak pri dekompozícií na r_i musí podmienka 3NF platiť pre každú r_i a nie pre celú r (intuícia hovorí, že pri uvažovaní celej r by to bola blbosť). Plachetka na ďalších slide-och potvrdzuje.

Boyce-Coddova normálna forma (BCNF). (Plachetka) Je v 2NF a pre každú platnú funkčnú závislosť $X \to Y$ platí, že X je nadkľúč (Wikipédia dodáva, že $\neg(X \supseteq Y)$.

Platí, že BCNF je prísnejšie ako 3NF (a zvykne sa preto označovať aj 3.5NF). Konštrukcia bezstratovej dekompozície do BCNF je NP ťažké, takisto, ako overenie jej existencie.

Príklad 3NF a nie BCNF. Ak máme reláciu ABC a funkčné závislosti $AB \to C$ a $C \to B$, tak je v 3NF (AB je kľúč) ale nie v BCNF, lebo C nie je kľúč v r. Rozloženie na (AC) a (BC) (podľa naivného algoritmu) pomôže (bezstratovosť: AC je kľúč).

7.2 Algoritmy pre dekompozíciu do normálnych foriem

Naivný 3NF. Nájdi nevyhovujúcu funkčnú závislosť (exponenciálne), minimalizuj ju na $X \to Y$ a rozlož príslušnú reláciu na relácie $r - Y$ a XY.

Polynomiálny 3NF. Nájdi minimálne pokrytie funkčnými závislosťami (polynomiálne). Pre každú $X \to Y$ z nich vytvor reláciu XY. Ak táto dekompozícia je stratová, pridaj reláciu pre ľubovoľný kľúč (bezstratovosť je ekvivalentná s existenciou takej tabuľky).

Naivný BCNF. Nájdi nevyhovujúcu funkčnú závislosť (exponenciálne), minimalizuj ju na $X \to Y$ a rozlož príslušnú reláciu na relácie $r - Y$ a XY.

7.3 Bezstratovosť dekompozície

(Názor autora, nesúhlasí s algoritmom v slide-och).

Algoritmus. Relačná schéma (r, F) s atribútmi $U_1, \ldots, U_n$ a s dekompozíciou $(r_1, F_1), \ldots, (r_m, F_m)$.

1. Vytvor maticu $S[i, j] = 1$ ak $U_j \in r_i$, inak $S[i, j] = 0$.
2. Zvyšok opakuj, kým sa niečo mení:
3. Pre každý $(X \to Y) \in F$:
4. Ak pre riadok r_i, platí $(\forall U_j \in X)S[i, j] = 1$, tak nastav $(\forall U_j)S[i, j] = 1$.

Ak ľubovoľný riadok pozostáva zo samých jednotiek, tak je bezstratová. Všimnime si, že riadky sú medzi sebou nezávislé a reprezentujú atribúty, ktoré sa dajú odvodiť z atribútov relácií r_i. Inak povedané, dekompozícia je bezstratová, ak existuje relácia, z ktorej atrbútov sa dá odvodiť funkčnými závislosťami ostatné atribúty v U.

Diskusia.

8 Transakcie

Predpokladáme centralizované databázy. Transakcií môže bežať naraz viac (a to je problém, ktorý chceme riešiť).

Operácie.

- START.
- READ / WRITE.
- INSERT / DELETE.
- COMMIT. Úspech transakcie.
- ABORT. Zrušenie transakcie, napríklad v prípade výpadku (alebo zlého schedulingu).

Skracujeme na {Prvé písmeno operácie}{Číslo transakcie}(argument). Napríklad $r1(A)$ znamená, že transakcia 1 číta pole A.

8.1 Požiadavky na transakčný systém (ACID)

- **A**tomicity. Všetko alebo nič. Buď transakcia prebehne, alebo nijakým spôsobom neovplyvní stav systému.
- **C**onsistency. Transakcia je operáciou z konzistentného stavu do konzistentného stavu (uzavretosť operácie).
- **I**solation. Vysledok transakcií je ekvivalentný s takým výsledkom, ktorý by sa dosiahol sériovým (postupným) vykonaním transakcií.
- **D**urability. Po príkaze COMMIT je transakcia platná navždy.

Tieto vlastnosti musia byť zachované za každých okolností, aj v prípade náhleho výpadku databázy alebo klienta.

8.2 Architektúra transakčného systému

Popisujeme len abstraktný model.

- Transaction manager. Bufferuje.
- Scheduler. Vytvára rozvrhy - poradia transakcií a ich jendotlivých príkazov.
- Recovery manager. Pamätá si log file, je schopný vykonať spätné zmeny (v prípade ABORTu napr.).
- Cache manager. Stará sa o to, aby časť disku bola v operačnej pamäti.

8.3 Rozvrhy

Rozvrh je postupnosť operácií transakcií, ktorá obsahuje operácie jednotlivých transakcií ako svoje podpostupnosti (ktoré sa neprekrývajú a rozvrh nič iné neobsahuje). *Aktívna transakcia*, ak je medzi START a COMMIT / ABORT.

Príklad zlého rozvrhu.

T1	T2
`Read(A); A <-- A+100`	
`Write(A);`	
	`Read(A);A <-- Ax2;`
	`Write(A);`
	`Read(B);B <-- Bx2;`
	`Write(B);`
	`Commit;`
`Read(B); B <-- B+100;`	
`Write(B);`	
`Commit;`	

8.4 Triedy sériovateľnosti a obnoviteľnosti

Konfliktné operácie sú dve operácie, ktoré patria dvom rôznym transakciám a aspoň jedna z nich je WRITE.

Históriou rozvrhu označujeme postupnosť operácií.

Konflikt-ekvivalentné sú dve histórie, keď majú rovnaké operácie a poradie konfliktných operácií je rovnaké v oboch históriách.

Transakcia $T2$ *číta* X od transakcie $T1$ práve vtedy, keď existuje $w1(X)$ pred $r2(X)$, takéto operácie nazveme *nepekné*.

View-ekvivalentné sú dve histórie $H1$ a $H2$, ak

- pre každú nepeknú dvojicu v $H1$ existuje rovnaká nepekná dvojica v $H2$ a
- a pre každý dátový objekt X je posledný WRITE v rovnakých transakciách[8].

Intuitívne sú dva rozvrhy view-ekvivalentné práve vtedy, keď majú rovnaký výsledok.

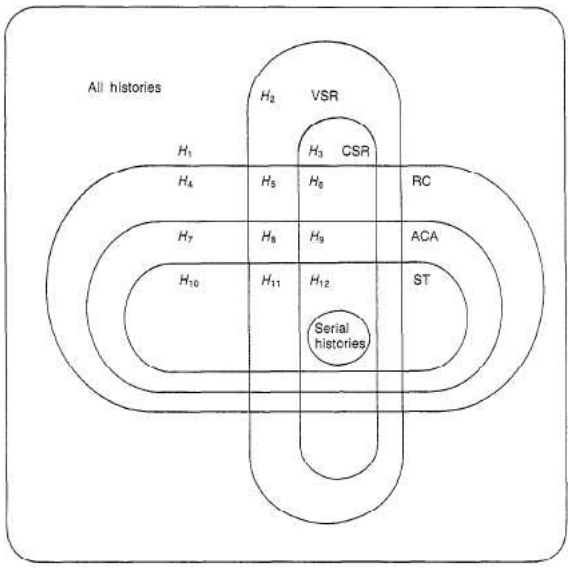
[8] Podmienka je nutná pre rozvrhy, ktoré majú viacero WRITEov na konci - také samozrejme nie sú serializovateľné

Triedy sériovateľnosti.

- *Sériové.* Žiadne dve transakcie sa neprekrývajú. Absolútne bezpečené, ibaže neefektívne (rýchle transakcie môžu čakať na jednu dlhú).
- *Sériovateľné (Konflikt sériovateľné).* Ak je konflikt-ekvivalentný nejakému sériovému rozvrhu.
- *View-sériovateľné.* Ak každý jeho prefix[9] je view-ekvivalentný nejakému sériovému rozvrhu. Rôzne poradie potenciálnych WRITEov nám nevadí (premyslite si).
- *Striktný rozvrh.* Ak rozvrh neobsahuje dirty read ani dirty write (čítanie a prepisovanie necommitovaných WRITEov).

Je zrejmé, že sériové $\subsetneq$ konflikt sériovateľné $\subsetneq$ view-sériovateľné. Viac v nasledujúcej sekcií 9.1.

Zemiak.



9 Implementácia sériovateľnosti a obnoviteľnosti v transakčných systémoch

Táto sekcia má svoje nedostatky.

9.1 Testy sériovateľnosti

Precedenčný graf rozvrhu je orientovaný graf na vrcholoch prislúchajúcich transakciám, pričom hrana z T_i do T_j existuje práve vtedy, keď O_i a O_j sú konfliktné operácie a O_i je v rozvrhu pred O_j. Podobným spôsobom vieme na základe relácie *číta* z view-sériovateľnosti definovať precedenčný graf pre view-sériovateľné rozvrhy.

Veta. Rozvrh je (konflikt) sériovateľný práve vtedy, keď jeho precedenčný graf je acyklický. Každé jeho topologické (čiastočné) usporiadanie (vrcholov=transakcií) je s ním konflikt-ekvivaletné.

Ale test view-sériovateľnosti je NP-ťažký problém[10].

[9] Ak niekto vie, prečo každý jeho prefix, a nie len celý rozvrh, tak mi prosím napíšte email.

[10] Rozmýšľam, prečo nestačí použiť rovnakú metódu precedenčného grafu.

9.2 Algoritmy izolácie, zámky, časové pečiatky, validácia

Áno, vieme generovať sériovateľné rozvrhy. Snažíme sa *izolovať* konflikty.

Zamykanie. Môžeme zamykať polia, riadky, tabuľky alebo celý diskový blok. Dva typy lockov:

- Read lock. Transakcia, ktorá ho vlastní, má právo čítať. Viacero transakcií môže mať read locky na rovnaké dáta.
- Write lock (Exkluzívny lock). Ak má transakcia write lock, tak žiadna iná transakcia nemôže mať žiadny iný lock na rovnaké dáta.

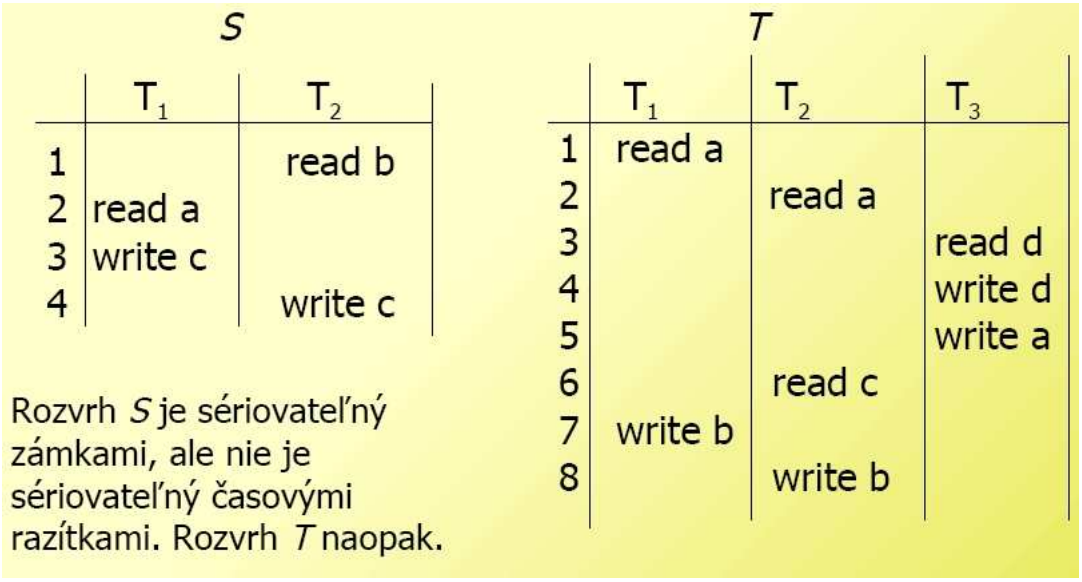
A dve základné pravidlá *dvojfázového zamykania*:

- Transakcia po unlocku už nemôže žiaden lock. Takže na začiatku si vypýta potrebné locky a na konci ich uvoľní.
- Transakcia musí vlastniť potrebný zámok na vykonanie príslušných operácií.

Scheduler kontroluje dodržanie týchto pravidiel, inak dáva priestor aktívnym transakciám prakticky akokoľvek. (Veta) Každý rozvrh vykonaný podľa lockov je konflikt-sériovateľný (dôkaz je jednoduchý - naopak to neplatí, napr $r1(X), r2(X), r1(X), w2(X)$).

Časové pečiatky. Idea je, kto prv príde, ten prv vykonáva. Každej transakcií je priradený čas začiatku TS_i a každý dátový objekt X má dve časové pečiatky, na čítanie TR_X a zápis TW_X (inicializujú sa na nulu). Ak T_i vykoná operáciu, do príslušnej časovej pečiatky zapíše TS_i. Dodržujú sa pritom dve pravidlá:

- Transakcia T_i nesmie čítať ak $TW_X > TS_i$.
- Transakcia T_i nesmie písať ak $TR_X > TS_i \vee TW_X > TS_i$.



Validácia (optimizmus). Každá transakcia si píše lokálne. Ak príde na COMMIT, tak sa skontroluje, či je doterajší rozvrh sériovateľný. Ak nie, tak ABORT na transakciu a všetky kolidujúce (reflexívno tranzitívny uzáver).

Multiversion concurrency control (MVCC) (optimizmus). Ku každému objektu si pamätáme viacere verzií podľa časov modifikácie. Podobne ako v algoritme časových pečiatok, aj tu definujeme TS_i - čas začiatku transakcie, TR_X a TW_X ako čas začatia transakcie, ktorá X čítala resp. písala. Obvyklé dve pravidlá:

- Ak transakcia T_i číta z X, tak dostane hodnotu v najstaršom čase pre ktorý $TW_X < TS_i$ a nastaví $TR_X = max(TR_X, TS_i)$.
- Ak transakcia T_i píše do X, tak je ABORTnutá v prípade, že $TW_X < TS_i < TR_X$ (lebo nejaká transakcia už prečítala, ale ešte nezapísala), inak $TW_X = max(TW_X, TS_i)$.

Strict 2PL. Rozširuje dvojfázové zamykanie o pravidlo, že zámky sa uvoľujú spolu s commitom. Takéto rozvrhy sú konflikt-sériovateľné a striktné.

9.3 Uviaznutie (deadlock) a metódy riešenia uviaznutia

Napríklad: $wl1(X), w1(X), rl2(Y), rl2(X), wl1(Y)$.

Wait-for graf - orientovaný graf nad reláciou T_i čaká na T_j (kvôli locku). Ak obsahuje cyklus, tak v deadlocku sú práve tie transakcie, ktoré sú na danom cykle.

Riešenia.

- Optimistická stratégia. Raz za čas skontrolujeme Wait-for graf. Ak cyklus, tak rozbijeme abortom.
- Pesimistická stratégia. Vopred detekujeme potenciálny deadlock a rozbijáme abortom.
- Wait-die stratégia. Transakcie sa usporiadajú podľa času začiatku. Ak skoršia žiada o lock staršiu, tak čaká na signál, kým staršia skončí.
- Wound-wait (kill-wait) stratégia. Transakcie sa usporiadajú podľa času začiatku. Ak skoršia žiada o lock staršiu, tak staršiu zabije (abortuje). Inak čaká na skončenie staršej.

9.4 Algoritmy obnovy, log-file, checkpointing, backup

Problém sériovateľného rozvrhu $r1(A), w1(A), r2(A), w2(C), c2, c1$ a výpadok pri $c2$. Potrebujeme preto obnovu a to potrebujeme LOG operácií (treba pritom dávať pozor na cache - nemôže sa zapisovať na disk aj cache zároveň).

Backup (DUMP) - zálohuje databázu na iné médium (počas toho neprebiehaújú žiadne transakcie).

- Okamžitý zápis dát. Najprv do logu a hneď potom na disk.
- Oneskorený zápis dát. Predpokladá sa cache, na disk sa zapisuje až niekedy po commite. Prečo netreba UNDO?

Logovanie. Stačí nám zapisovať iba WRITE operácie, so starou aj novou hodnotou (aby sme vedeli overiť, či sa naozaj vykonala). Raz za čas môže prebehnúť CHECKPOINTING (atomická procedúra, myslí sa na aktívne transakcie) - overenie, že sa po istý riadok LOG file-u vykonali všetky operácie.

Obnova. Čítaním log file-u sa vytvára UNDO a REDO list.

- UNDO vráti neCOMMITované transakcie (v opačnom poradí). Stav po UNDO je rovnaký, ako keby sa operácie nevykonali.
- REDO vykoná naspäť UNDO operácie a navráti pôvodný stav.

Abort. Rozvrh je *obnoviteľný* práve vtedy, keď pre každú commmitovanú transakciu T2, ktorá čítala necommitovanú hodnotu zapísanú T1 (*dirty read*), commituje T1 skôr ako T2.

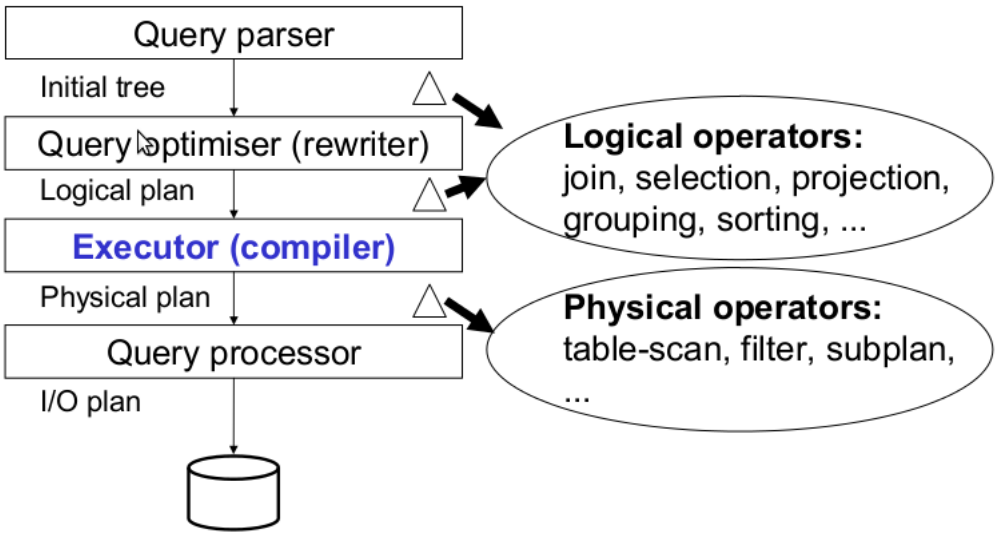
Príklady:

- Neobnoviteľný: $w1(X), r2(X), w2(X), c2, c1$.
- Obnoviteľný: $w1(X), r2(X), w2(X), c1, c2$. Vzniká riziko kaskádového abortu.

Kaskádový abort. Keď abort jednej transakcie vynúti abort inej transakcie (kvôli dirty read). *Avoids cascading aborts (ACA)* práve vtedy, keď rozvrh neobsahuje žiadny dirty read.

10 Fyzická organizácia

Prerekvizity: Operačné systémy - organizácia a prístup k dátam. Cieľom je minimalizovať počet diskovýh operácií (sú rádovo 100krát pomalšie ako RAM) a byť schopný pracovať aj s tabuľkami väčšími ako veľkosť operačnej pamäte.



Na popis, ako funguje query, slúži príkaz `EXPLAIN`. Dá sa pomocou neho značne zefektívniť beh dotazu.

Označenie.

- $B(R)$: počet blokov R
- $T(R)$: počet tuples (záznamov) R, vrátane duplikátov
- $V(R, a_1, a_2, ..., a_N)$: počet rôznych tuples $\Pi_{a1,a2,...,a_N}(R)$

Medzivýsledky.

- Materializácia - môže zaberať veľký priestor.
- Iterátory - namiesto materializácie sa vytvára iterátor, ktorý prechádza tabuľkou. Majú tri funkcie:
  - open(R).
  - next(R).
  - close(R).

10.1 Dvojúrovňový model pamäti a organizácie dát

Operačná pamäť (RAM) a disk. Snažíme sa redukovať počet diskových čítaní. Pričom vieme, že zaberie rovnako času čítanie jedného celého bloku (sektoru) ako jeho ľubovoľnej časti. Poznamenajme, že rýchlosť operačnej pamäte je rádovo stonásobne vyššia a jej nevýhodou je, že sa do nej nemusia zmestiť celé tabuľky.

10.2 Indexové stromy, hashovanie

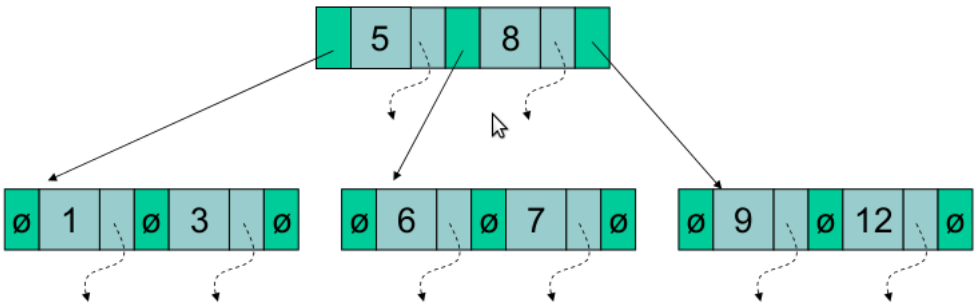
Jednou možnosťou je mať tabuľku vždy utriedenú. Oveľa jednoduchšie a praktickejšie je mať externý index v nezávislom súbore - ISAM (Index Sequential Access Method). Kľúče, podľa ktorých sa usporiadava, nemusia nijako súvisieť s tými z funkčných závislosti (aj keď PRIMARY INDEX by mal byť kľúčom relácie).

Sekvenčné indexy.

- Dense. Referencia na každý riadok tabuľky.
- Sparse. Referencia len na bloky tabuliek (ktoré sú ďalej usporiadané). Optimalizáciou je v poslednom bloku si pamätať referenciu na nasledujúci (continuous). Názorné príklady v slide-och.
- Multilevel. Index indexu. Najvnútornejší je vždy dense.

Výhodou je rýchle sekvenčné prehľadávanie. V prípade hustého indexu vieme rozhodnúť o existencií záznamu bez čítania celého záznamu. Nevýhodou je drahé vkladanie a hlavne vyhľadávanie.

B stromy. *B(M) stromy* sú podobné, ako binárne vyhľadávacie stromy, len sa v každom vnútornom (nelistovom) vrchole vetvia maximálne m krát a minimálne $\lfloor m/2 \rfloor$ krát. Ak sa vrchol vetví k krát, tak sa v ňom pamätá $k-1$ záznamov, ktorými sa oddeľujú hodnoty v podstromoch. Napríklad 2-3 strom (štátnicová otázka) je B(3) stromom.



B^+ stromy. Majú vo vnútorných uzloch (nie listoch) len kľúče - index (resp. ich hodnoty). Dátové záznamy sú iba v listoch. Vkladanie a vyberanie je celkom intuitívne (treba rozobrať viacero prípadov, aby sa zachovala podmienka $\lfloor m/2 \rfloor \leq k \leq m$, kde k je stupeň vrchola). Prakticky nastáva buď triviálny prípad, alebo sa to rieši rekurzívne. Názorné príklady v slide-och.

- Výhody. Kvôli veľkému vetveniu vyžaduje relatívne málo idskových operácií. Vhodné na prehľadávanie, vyhľadávanie aj mazanie (väčsinou triviálny prípad).
- Nevýhody. Pamäťový overhead. Netriviálna implementácia.

Hashovanie. Princíp hashovania by mal byť čitateľovi zrejmý. Hľadáme *ideálnu hashovaciu funkciu*, ktorá nám hashe rozdelí na rovnomerné buchety a nebude prichádzať k preplneniu žiadneho z nich. Jednotlivé buckety chceme mať v rovnakých diskových blokoch (aj preto spájané zoznamy prislúchajúce k rovnakému hashu rozdeľujeme na bloky).

Magické konštanty pre *load factor* $\frac{vyuzitych\_blokov}{vsetkych\_blokov}$ sú 50% a 80%.

- Priama adresácia. Máme pole s riadkami tabuľky.
- Neriama adresácia. Máme pole, v ktorých sú referencie na riadky tabuľky.

Rozšíriteľné hashovanie. Používame iba prvých i bitov hashu, ak nastane preplnenie poľa, tak zvýšime i (adresný priestor sa teda zdvojnásobí).

Lineárne hashovanie . Používame hashovaciu funkciu $h_{p,q}(K) = Kmod(2^q \cdot p)$. Pamätáme si pointer n na blok, počet blokov M a počet prvkov N. Ak sa presiahne load factor, tak sa pridá nový blok, rozdelením bloku n a zvýšením $q++$ pre tento blok, posunutím pointra $n++$ a zvýšením počtu blokov $M++$ (ak $M = 2^x$, tak $n = 0$). Na bloky $0 \leq k < n$ a $2^q \cdot p$ používame teda $h_{p,q+1}$ a na ostatné $h_{p,1}$. Veľmi pomôže video. Vyhľadávanie bloku:

```
n=N;
k=K mod N; /* hashovacia funkcia */
while ((k>p) and (not A[k]))
{
n = n / 2; /* celočíselné delenie */
k = k { n;
}
```

Porovnanie s B stromami.

- Výhody. Teoreticky lineárny čas, celková jednoduchosť.
- Nevýhody. B stromy lepšie vracajú intervaly výsledkov.

10.3 Operátory fyzickej algebry

???

10.4 Implementácia vybraných fyzických operátorov (merge-sort, nested-loop join)

Mergesort. Chceme utriediť pole dĺžky N a do operačnej pamäti sa nám zmestí $3B$ prvkov. V prvej iterácií načítame bloky dĺžky B, utriedime v operačnej pamäti a zapíšeme na disk. Keď potom mergeujeme dva, vstupné bloky dĺžok D, tak načítame z oboch B a mergeujeme do tretieho - výstupného (v operačnej pamäti máme $3B$). Ak sa vyprázdni vstupný, tak načítame ďalších B prvkov. Ak sa naplní výstupný, zapíšeme na disk a vyprázdnime.

Zložitosť vie vypočítať každý z vzhľadom na cenu diskovej operácie a počtu krokov.

Join. Podobne ako mergesort. Ak sa do operačnej pamäti zmestí M diskových blokov, tak $M-2$ vyhradíme na menšiu z relácií, 1 na druhú reláciu a jeden na výstup.

Merge Join (equijoin). Predpokladáme utriedené joinovacie relácie a joinovaciu podmienku pre rovnosť. Potom vieme join vypočítať podobným spôsobom ako mergesort, akurát s a potrebujeme zbaviť duplikátov.

Index scan. Čítanie viacerých za sebou idúcich záznamov. Nájdi prvý vyhovujúci záznam, ďalšie stačí hľadať v nasledujúcich blokoch. `cost = height + b + 1`, kde `height` je hĺbka indexového stromu a `b` je počet blokov obsahujúcich vyhovujúce záznamy.

Hash join. Máme dva hash-indexy nad reláciami. Chceme ich joinovať s rovnosťou. Joinovacia pdomienka bude splnená len pre rovnaké hashe, stačí nám čítať príslušné buckety (bloky) a ich prípadné buckety.

Referencie a odporúčaná literatúra

- Úvodu do databázových systémov - Plachetka.
- Úvodu do databázových systémov - Šturc - v niečom detailnejší.
- Úvodu do databázových systémov - Stanford University - základ podobný.
- Plachetkove slide-i.
- Ullmanove slide-i.
- Mandos.
- H. Garcia-Molina, J.D. Ullman, J. Widom: Database Systems, The Complete Book, Prentice Hall, 2003
- R. Elmasri, S.B. Navathe: Fundamentals of Database Systems, Addison-Wesley, 2006
- Na poslednú otázku[11]: S. Lightstone, T.J. Teorey, T. Nadeau: Physical Database Design, Morgan Kaufmann, 2007

[11] Na ktorej už zopár ľudí dostalo Fx.